单例模式（Singleton Pattern）

确保一个类只有一个实例，并提供该实例的全局访问点。

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。

优点：

- 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。
- 2、避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

使用场景：

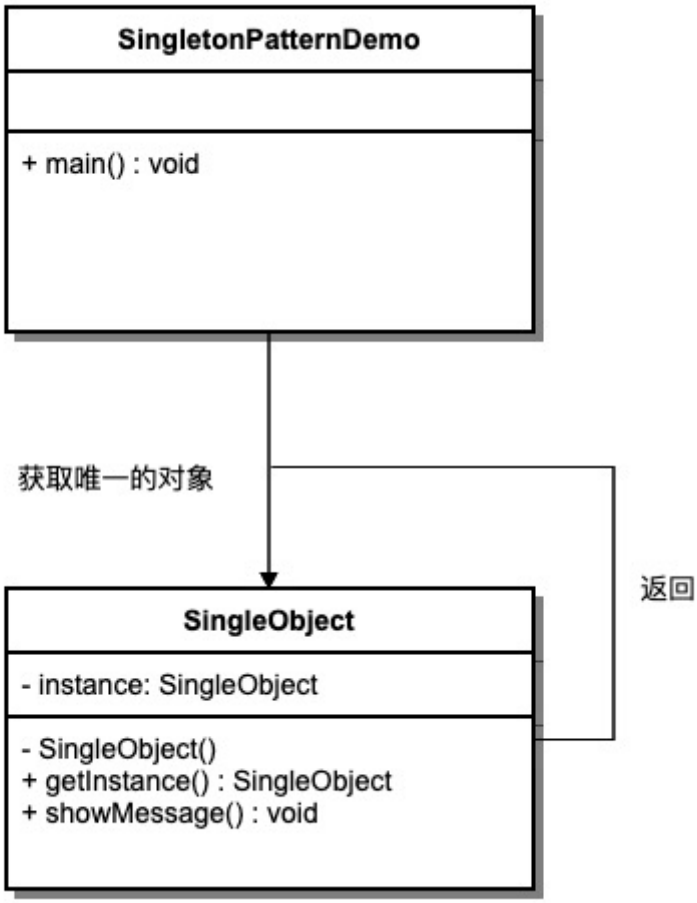
- 1、要求生产唯一序列号。
- 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。
- 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

注意事项：getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

实现

使用一个私有构造函数、一个私有静态变量以及一个公有静态函数来实现。

私有构造函数保证了不能通过构造函数来创建对象实例，只能通过公有静态函数返回唯一的私有静态变量。



1、懒汉式，线程不安全

描述：使用延迟加载（lazy loading），线程不安全。这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){
    }

    public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

接下来介绍的几种实现方式都支持多线程，但是在性能上有所差异。

2、懒汉式，线程安全

使用延迟加载（lazy loading），线程安全。只需要对 getUniqueInstance() 方法加锁，那么在一个时间点只能有一个线程能够进入该方法，从而避免了实例化多次 uniqueInstance。

但是当一个线程进入该方法之后，其它试图进入该方法的线程都必须等待，即使 uniqueInstance 已经被实例化了。这会让线程阻塞时间过长，因此该方法有性能问题，不推荐使用。

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
    public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

3、饿汉式

是否 Lazy 初始化：否

是否多线程安全：是

描述：这种方式比较常用，但容易产生垃圾对象。
优点：没有加锁，执行效率会提高。
缺点：类加载时就初始化，浪费内存。
它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
        return instance;
    }
}
```

4、双检锁/双重校验锁（DCL，即 double-checked locking)

JDK 版本：JDK1.5 起

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：较复杂

描述：这种方式采用双锁机制，安全且在多线程情况下能保持高性能。getInstance() 的性能对应用程序很关键。

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton (){
    }
    public static Singleton getSingleton() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
```

```
            return singleton;
        }
    }
```

考虑下面的实现，也就是只使用了一个 if 语句。在 singleton== null 的情况下，如果两个线程都执行了 if 语句，那么两个线程都会进入 if 语句块内。虽然在 if 语句块内有加锁操作，但是两个线程都会执行 `singleton= new Singleton();` 这条语句，只是先后的问题，那么就会进行两次实例化。因此必须使用双重校验锁，也就是需要使用两个 if 语句：第一个 if 语句用来避免 singleton已经被实例化之后的加锁操作，而第二个 if 语句进行了加锁，所以只能有一个线程进入，就不会出现 singleton== null 时两个线程同时进行实例化操作。

```
if (singleton == null) {
    synchronized (Singleton.class) {
        singleton = new Singleton();
    }
}
```

singleton采用 volatile 关键字修饰也是很有必要的， `uniqueInstance = new Singleton();` 这段代码其实是分为三步执行：

1. 为 singleton分配内存空间
2. 初始化 singleton
3. 将 singleton指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1>3>2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getSingleton() 后发现 singleton不为空，因此返回 singleton，但此时 singleton还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

5、登记式/静态内部类

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：一般

描述：这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。
这种方式同样利用了 classloader 机制来保证初始化 instance 时只有一个线程，它跟第 3 种方式不同的是：第 3 种方式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有通过显式调用 getInstance 方法时，才会显式装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第 3 种方式就显得很合理。

```
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

6、枚举

JDK 版本：JDK1.5 起

是否 Lazy 初始化：否

是否多线程安全：是

实现难度：易

描述：这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。
这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还自动支持序列化机制，防止反序列化重新创建新的对象，绝对防止多次实例化。不过，由于 JDK1.5 之后才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，也很少用。
不能通过 reflection attack 来调用私有构造方法。

```
public enum Singleton {
    INSTANCE;

    private String objName;


    public String getObjName() {
        return objName;
    }

    public void setObjName(String objName) {
        this.objName = objName;
    }

    public static void main(String[] args) {
        // 单例测试
        Singleton firstSingleton = Singleton.INSTANCE;
        firstSingleton.setObjName("firstName");
        System.out.println(firstSingleton.getObjName());
        Singleton secondSingleton = Singleton.INSTANCE;
        secondSingleton.setObjName("secondName");
        System.out.println(firstSingleton.getObjName());
        System.out.println(secondSingleton.getObjName());

        // 反射获取实例测试
        try {
            Singleton[] enumConstants = Singleton.class.getEnumConstants();
            for (Singleton enumConstant : enumConstants) {
                System.out.println(enumConstant.getObjName());
            }
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

该实现可以防止反射攻击。在其它实现中，通过 setAccessible() 方法可以将私有构造函数的访问级别设置为 public，然后调用构造函数从而实例化对象，如果要防止这种攻击，需要在构造函数中添加防止多次实例化的代码。该实现是由 JVM 保证只会实例化一次，因此不会出现上述的反射攻击。

该实现在多次序列化和序列化之后，不会得到多个实例。而其它实现需要使用 transient 修饰所有字段，并且实现序列化和反序列化的方法。

经验之谈：一般情况下，不建议使用第 1 种和第 2 种懒汉方式，建议使用第 3 种饿汉方式。只有在要明确实现 lazy loading 效果时，才会使用第 5 种登记方式。如果涉及到反序列化创建对象时，可以尝试使用第 6 种枚举方式。如果有其他特殊的需求，可以考虑使用第 4 种双检锁方式。

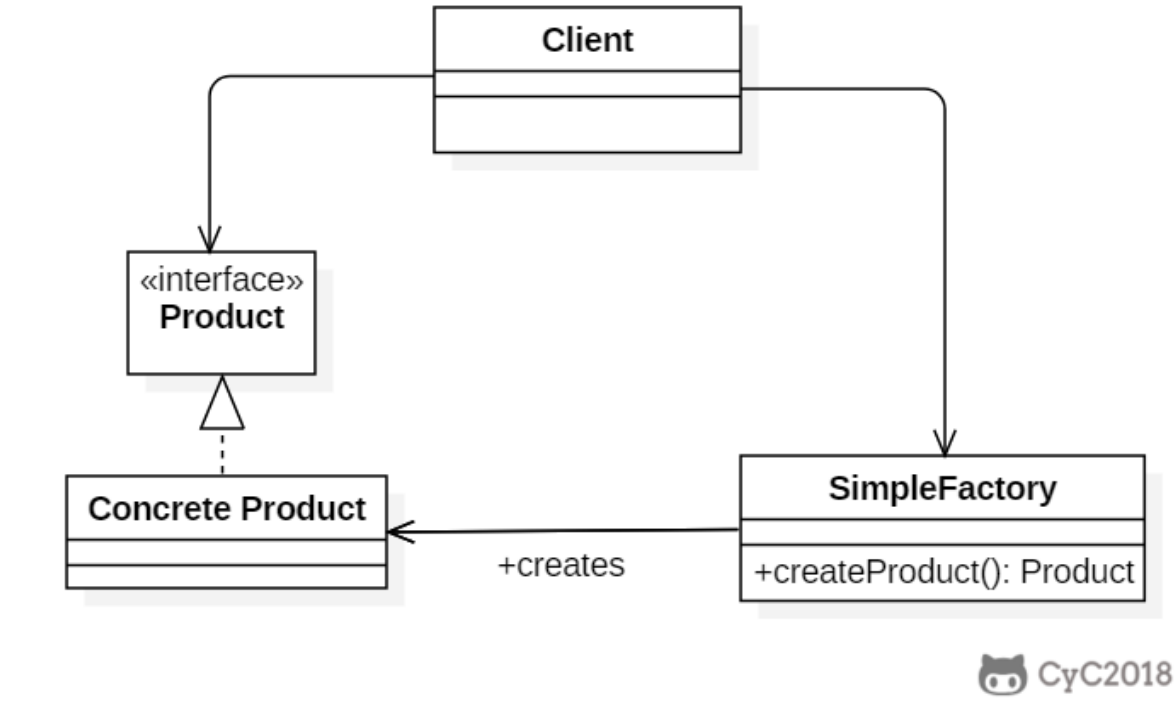
简单工厂（Simple Factory）

在创建一个对象时不向客户暴露内部细节，并提供一个创建对象的通用接口。

Class Diagram

简单工厂把实例化的操作单独放到一个类中，这个类就成为简单工厂类，让简单工厂类来决定应该用哪个具体子类来实例化。

这样做能把客户类和具体子类的实现解耦，客户类不再需要知道有哪些子类以及应当实例化哪个子类。客户类往往有多个，如果不使用简单工厂，那么所有的客户类都要知道所有子类的细节。而且一旦子类发生改变，例如增加子类，那么所有的客户类都要进行修改。



Implementation

1.创建一个接口

2.创建多个实现接口的实体类。

3.创建一个工厂，包含一个方法，该方法基于给定参数生成对应的的实体类对象。

4.使用该工厂，通过传递类型信息来获取实体类的对象。

```
//1.创建一个接口
interface Product {
}

//2.创建3个实现接口的实体类
class ConcreteProduct implements Product {
}

class ConcreteProduct1 implements Product {
}

class ConcreteProduct2 implements Product {
}
//3.简单工厂的实现，它被所有需要进行实例化的客户类调用。
class SimpleFactory {

    public Product createProduct(int type) {
        if (type == 1) {
            return new ConcreteProduct1();
        } else if (type == 2) {
            return new ConcreteProduct2();
        }
        return new ConcreteProduct();
    }
}

//4.使用工厂
public class Client {

    public static void main(String[] args) {
        SimpleFactory simpleFactory = new SimpleFactory();
        Product product = simpleFactory.createProduct(1);
        // do something with the product
    }
}
```

以下的 Client 类包含了实例化的代码，这是一种错误的实现。如果在客户类中存在这种实例化代码，就需要考虑将代码放到简单工厂中。

```
public class Client {

    public static void main(String[] args) {
        int type = 1;
        Product product;
        if (type == 1) {
            product = new ConcreteProduct1();
        } else if (type == 2) {
```

```
            product = new ConcreteProduct2();
        } else {
            product = new ConcreteProduct();
        }
        // do something with the product
    }
}
```

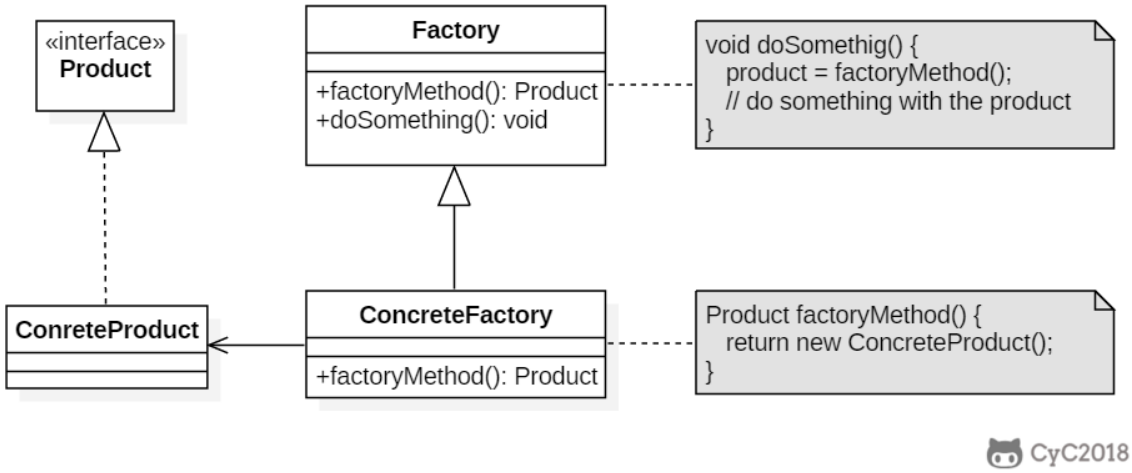
工厂方法（Factory Method）

定义了一个创建对象的接口，但由子类决定要实例化哪个类。工厂方法把实例化操作推迟到子类。

Class Diagram

在简单工厂中，创建对象的是另一个类，而在工厂方法中，是由子类来创建对象。

下图中，Factory 有一个 doSomething() 方法，这个方法需要用到一个产品对象，这个产品对象由 factoryMethod() 方法创建。该方法是抽象的，需要由子类去实现。



Implementation

1.创建接口(抽象类)

2.创建接口的具体实现类

3.创建接口对应的工厂

4.每一个接口对应的实现类都创建一个与其对应的工厂实现类

```
public abstract class Factory {
    abstract public Product factoryMethod();
    public void doSomething() {
        Product product = factoryMethod();
        // do something with the product
    }
}
```

```
public class ConcreteFactory extends Factory {
    public Product factoryMethod() {
        return new ConcreteProduct();
    }
}
```

```
public class ConcreteFactory1 extends Factory {
    public Product factoryMethod() {
        return new ConcreteProduct1();
    }
}
```

```
public class ConcreteFactory2 extends Factory {
    public Product factoryMethod() {
        return new ConcreteProduct2();
    }
}
```

```
public class Client{
    public static void main(String[] ags){
        Factory cf1=new ConcreteFactory().factoryMethod();
        Factory cf2=new ConcreteFactory1().factoryMethod();
        Factory cf3=new ConcreteFactory2().factoryMethod();
    }
}
```

4抽象工厂（Abstract Factory）

提供一个接口，用于创建**相关的对象家族** 。

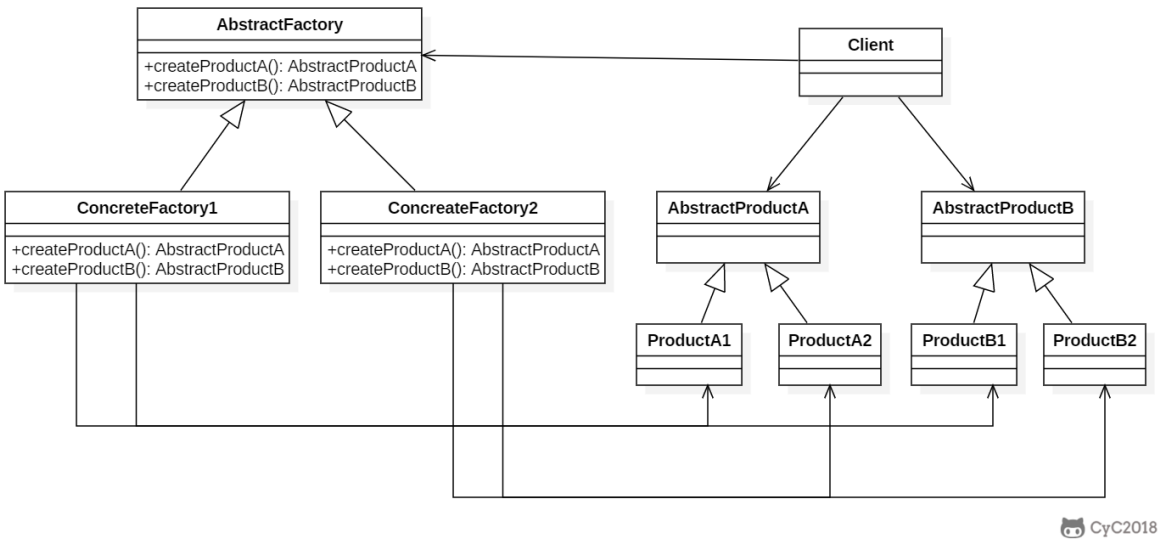
Class Diagram

抽象工厂模式创建的是对象家族，也就是很多对象而不是一个对象，并且这些对象是相关的，也就是说必须一起创建出来。而工厂方法模式只是用于创建一个对象，这和抽象工厂模式有很大不同。

抽象工厂模式用到了工厂方法模式来创建单一对象，AbstractFactory 中的 createProductA() 和 createProductB() 方法都是让子类来实现，这两个方法单独来看就是在创建一个对象，这符合工厂方法模式的定义。

至于创建对象的家族这一概念是在 Client 体现，Client 要通过 AbstractFactory 同时调用两个方法来创建出两个对象，在这里这两个对象就有很大的相关性，Client 需要同时创建出这两个对象。

从高层次来看，抽象工厂使用了组合，即 Cilent 组合了 AbstractFactory，而工厂方法模式使用了继承。



Implementation

1.创建多个产品类（可以是普通类/接口/抽象类）

2.每一个产品类（普通类/接口/抽象类）对应生成多种不同的产品子类

3.创建工厂类（抽象类/接口），包括对每一个产品类的创建方法

4.创建多个工厂实现类，每个实现类实现不同的产品子类的组合

```
//1.创建多个产品类
class AbstractProductA {
}
class AbstractProductB {
}
```

```
//2.创建产品子类
class ProductA1 extends AbstractProductA {
}
class ProductA2 extends AbstractProductA {
}
class ProductB1 extends AbstractProductB {
}
class ProductB2 extends AbstractProductB {
}
```

```
//3.创建工厂类（抽象类/接口），包括对每一个产品类的创建方法
abstract class AbstractFactory {
    abstract AbstractProductA createProductA();
    abstract AbstractProductB createProductB();
}
```

```
//4.创建多个工厂实现类
ConcreteFactory1 extends AbstractFactory {
    AbstractProductA createProductA() {
        return new ProductA1();
    }
    AbstractProductB createProductB() {
        return new ProductB1();
    }
}
ConcreteFactory2 extends AbstractFactory {
    AbstractProductA createProductA() {
        return new ProductA2();
    }

    AbstractProductB createProductB() {
        return new ProductB2();
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        AbstractFactory abstractFactory = new ConcreteFactory1();
        AbstractProductA productA = abstractFactory.createProductA();
        AbstractProductB productB = abstractFactory.createProductB();
        // do something with productA and productB
    }
}
```

例二

```
//为形状创建一个接口。
public interface Shape {
    void draw();
}
//为颜色创建一个接口。
public interface Color {
   void fill();
}
```

```
//创建形状接口的实体类。
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}

//创建实现接口的实体类。
public class Red implements Color {

   @Override
   public void fill() {
      System.out.println("Inside Red::fill() method.");
   }
}
public class Green implements Color {

   @Override
   public void fill() {
      System.out.println("Inside Green::fill() method.");
   }
```

```
}
public class Blue implements Color {

   @Override
   public void fill() {
      System.out.println("Inside Blue::fill() method.");
   }
}
```

```
//为 Color 和 Shape 对象创建抽象类来获取工厂。
public abstract class AbstractFactory {
   public abstract Color getColor(String color);
   public abstract Shape getShape(String shape) ;
}
```

```
//创建扩展了 AbstractFactory 的形状工厂类，基于给定的信息生成实体类的对象。
public class ShapeFactory extends AbstractFactory {

   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }

   @Override
   public Color getColor(String color) {
      return null;
   }
}
//创建扩展了 AbstractFactory 的颜色工厂类，基于给定的信息生成实体类的对象。
public class ColorFactory extends AbstractFactory {

   @Override
   public Shape getShape(String shapeType){
      return null;
   }

   @Override
   public Color getColor(String color) {
      if(color == null){
         return null;
      }
      if(color.equalsIgnoreCase("RED")){
         return new Red();
      } else if(color.equalsIgnoreCase("GREEN")){
         return new Green();
      } else if(color.equalsIgnoreCase("BLUE")){
```

```
         return new Blue();
      }
      return null;
   }
}
```

```
//创建一个工厂创造器/生成器类，通过传递形状或颜色信息来获取工厂。
public class FactoryProducer {
   public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();
      } else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }
      return null;
   }
}
```

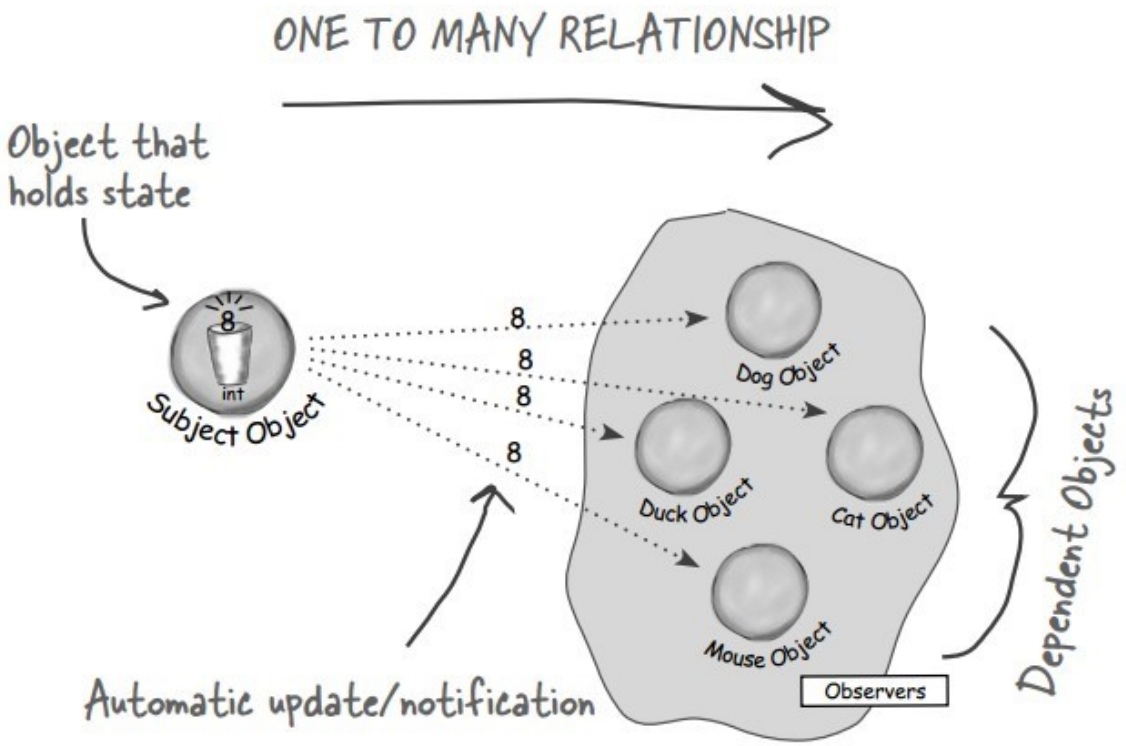
```
//使用 FactoryProducer 来获取 AbstractFactory，通过传递类型信息来获取实体类的对象。
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {

      //获取形状工厂
      AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

      //获取形状为 Circle 的对象
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //调用 Circle 的 draw 方法
      shape1.draw();

      //获取形状为 Rectangle 的对象
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //调用 Rectangle 的 draw 方法
      shape2.draw();

      //获取形状为 Square 的对象
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //调用 Square 的 draw 方法
      shape3.draw();

      //获取颜色工厂
      AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

      //获取颜色为 Red 的对象
      Color color1 = colorFactory.getColor("RED");

      //调用 Red 的 fill 方法
      color1.fill();

      //获取颜色为 Green 的对象
      Color color2 = colorFactory.getColor("Green");

      //调用 Green 的 fill 方法
```

```
        color2.fill();

        //获取颜色为 Blue 的对象
        Color color3 = colorFactory.getColor("BLUE");

        //调用 Blue 的 fill 方法
        color3.fill();
    }
}
```

观察者（Observer）

定义对象之间的一对多依赖，当一个对象状态改变时，它的所有依赖都会收到通知并且自动更新状态。

主题（Subject）是被观察的对象，而其所有依赖者（Observer）称为观察者。



Class Diagram

主题（Subject）具有注册和移除观察者、并通知所有观察者的功能，主题是通过维护一张观察者列表来实现这些操作的。

观察者（Observer）的注册功能需要调用主题的 registerObserver() 方法。

Implementation

1.创建主题类（类/接口）

2.创建观察者接口

3.创建主题类实现类，成员如下：

属性：

1.观察者列表

2.被观察的成员变量

方法：

1.注册、移除、通知所有观察者方法(notifyObserver会调用observe的update方法)

2.对被观察的成员变量进行操作，并调用通知所有观察者方法(notifyObserver)的方法

4.创建观察者实现类，包含一个对更新方法的实现

```
public interface Subject {
    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObserver();
}
```

```
public class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<>();
    }

    public void setMeasurements(float temperature, float humidity, float
pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        notifyObserver();
    }

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    @Override
    public void notifyObserver() {
        for (Observer o : observers) {
            o.update(temperature, humidity, pressure);
        }
    }
}
```

```java
public interface Observer {
    void update(float temp, float humidity, float pressure);
}
```

```java
public class StatisticsDisplay implements Observer {

    public StatisticsDisplay(Subject weatherData) {
        weatherData.reisterObserver(this);
    }

    @Override
    public void update(float temp, float humidity, float pressure) {
        System.out.println("StatisticsDisplay.update: " + temp + " " + humidity
+ " " + pressure);
    }
}
```

```java
public class CurrentConditionsDisplay implements Observer {

    public CurrentConditionsDisplay(Subject weatherData) {
        weatherData.registerObserver(this);
    }

    @Override
    public void update(float temp, float humidity, float pressure) {
        System.out.println("CurrentConditionsDisplay.update: " + temp + " " +
humidity + " " + pressure);
    }
}
```

```java
public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
        CurrentConditionsDisplay currentConditionsDisplay = new
CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new
StatisticsDisplay(weatherData);

        weatherData.setMeasurements(0, 0, 0);
        weatherData.setMeasurements(1, 1, 1);
    }
}
```

```java
import java.util.ArrayList;
import java.util.List;

public class client {
    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();
        ObserverA a = new ObserverA();
        ObserverA b = new ObserverA();
        ObserverA c = new ObserverA();
        //将这三个观察者添加到subject对象的观察者对象中
        subject.registerObserver(a);
```

```
        subject.registerObserver(b);
        subject.registerObserver(c);

        //改变subject的状态
        subject.setState(3000);
        System.out.println("**************************");
        //看看观察者的值有没有变化
        System.out.println(a.getMyState());
        System.out.println(b.getMyState());
        System.out.println(c.getMyState());
    }
}


class Subject {
    private List<Observer> list = new ArrayList<>();

    public void registerObserver(Observer obs) {
        list.add(obs);
    }

    public void removeObserver(Observer obs) {
        int i = list.indexOf(obs);

        if (i >= 0) {
            list.remove(i);
        }
    }

    public void notifyObserver() {
        for (Observer o : list) {
            o.update(this);
        }
    }
}


interface Observer {
    void update(Subject subject);
}


class ConcreteSubject extends Subject {
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        this.notifyObserver();
    }
}


class ObserverA implements Observer {
    private int myState;//myState需要跟目标对象的state值保持一致
```

```
    @Override
    public void update(Subject subject) {
        myState = ((ConcreteSubject) subject).getState();
    }

    public int getMyState() {
        return myState;
    }

    public void setMyState(int myState) {
        this.myState = myState;
    }
}
```

通过obserable类和observer接口实现观察者模式

1.具体主题继承Observable类，自定义方法调用notifyObservers

2.具体观察者实现Observer接口，重写update方法

```
import java.util.Observable;
import java.util.Observer;

public class client {
    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();
        ObserverA a = new ObserverA();
        ObserverA b = new ObserverA();
        ObserverA c = new ObserverA();
        //将这三个观察者添加到subject对象的观察者对象中
        subject.addObserver(a);
        subject.addObserver(b);
        subject.addObserver(c);

        //改变subject的状态
        subject.setState(3000);
        System.out.println("**************************");
        //看看观察者的值有没有变化
        System.out.println(a.getMyState());
        System.out.println(b.getMyState());
        System.out.println(c.getMyState());
    }
}


//目标对象
class ConcreteSubject extends Observable {
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;//目标对象的状态发生了改变
        setChanged();//表示目标对象已经做出了改变
```

```
        notifyObservers(this.state);//通知多有的观察者
    }
}


class ObserverA implements Observer {
    private int myState;//myState需要跟目标对象的state值保持一致



    public int getMyState() {
        return myState;
    }

    public void setMyState(int myState) {
        this.myState = myState;
    }

    @Override
    public void update(Observable o, Object arg) {
        myState = ((ConcreteSubject) o).getState();

    }
}
```

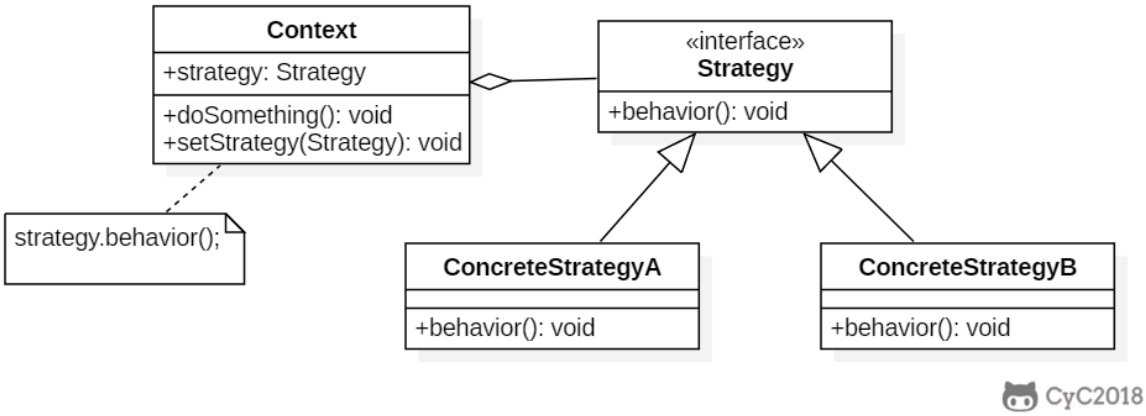
策略（Strategy）

定义一系列算法，封装每个算法，并使它们可以互换。

策略模式可以让算法独立于使用它的客户端。

Class Diagram

- Strategy 接口定义了一个算法族，它们都实现了 behavior() 方法。
- Context 是使用到该算法族的类，其中的 doSomething() 方法会调用 behavior()，setStrategy(Strategy) 方法可以动态地改变 strategy 对象，也就是说能动态地改变 Context 所使用的算法。



与状态模式的比较

状态模式的类图和策略模式类似，并且都是能够动态改变对象的行为。但是状态模式是通过状态转移来改变 Context 所组合的 State 对象，而策略模式是通过 Context 本身的决策来改变组合的 Strategy 对象。所谓的状态转移，是指 Context 在运行过程中由于一些条件发生改变而使得 State 对象发生改变，注意必须要是在运行过程中。

状态模式主要是用来解决状态转移的问题，当状态发生转移了，那么 Context 对象就会改变它的行为；而策略模式主要是用来封装一组可以互相替代的算法族，并且可以根据需要动态地去替换 Context 使用的算法。

Implementation

设计一个鸭子，它可以动态地改变叫声。这里的算法族是鸭子的叫声行为。

```
public interface QuackBehavior {
    void quack();
}
```

```
public class Quack implements QuackBehavior {
    @Override
    public void quack() {
        System.out.println("quack!");
    }
}
```

```
public class Squeak implements QuackBehavior{
    @Override
    public void quack() {
        System.out.println("squeak!");
    }
}
```

```
public class Duck {

    private QuackBehavior quackBehavior;

    public void performQuack() {
        if (quackBehavior != null) {
            quackBehavior.quack();
        }
    }

    public void setQuackBehavior(QuackBehavior quackBehavior) {
        this.quackBehavior = quackBehavior;
    }
}
```

```
public class Client {

    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.setQuackBehavior(new Squeak());
        duck.performQuack();
        duck.setQuackBehavior(new Quack());
        duck.performQuack();
    }
}
```

```
squeak!
quack!
```

不采用策略模式

```
/**
 * 实现起来比较容易，符合一般开发人员的思路
 * 假如，类型特别多，算法比较复杂时，整个条件语句的代码就变得很长，难于维护。
 * 如果有新增类型，就需要频繁的修改此处的代码!
 * 不符合开闭原则!
 */
class TestStrategy {
    public double getPrice(String type, double price) {
        if (type.equals("普通客户小批量")) {
            System.out.println("不打折,原价");
            return price;
        } else if (type.equals("普通客户大批里")) {
            System.out.println("打九折");
            return price * 0.9;
        } else if (type.equals("老客户小批童")) {
            System.out.println("打八五折");
            return price * 0.85;
        } else if (type.equals("老客户大批童")) {
            System.out.println("打八折");
            return price * 0.8;
        }
        return price;
    }
}
```

采用策略模式

```
public class client {
    public static void main(String[] args) {
        Strategy s1 = new OldCustomerFewStrategy();
        Context ctx = new Context(s1);
        ctx.printPrice(998);
    }
}




interface Strategy {
```

```
    public double getPrice(double standardPrice);
}

class NewCustomerFewStrategy implements Strategy {

    @Override
    public double getPrice(double standardPrice) {
        System.out.println("不打折，原价");
        return standardPrice;
    }
}


class NewCustomerManyStrategy implements Strategy {

    @Override
    public double getPrice(double standardPrice) {
        System.out.println("打九折");
        return standardPrice * 0.9;
    }
}

class OldCustomerFewStrategy implements Strategy {

    @Override
    public double getPrice(double standardPrice) {
        System.out.println("打八五折");
        return standardPrice * 0.85;
    }
}


class OldCustomerManyStrategy implements Strategy {

    @Override
    public double getPrice(double standardPrice) {
        System.out.println("打八折");
        return standardPrice * 0.8;
    }
}

/**
 * 负责和具体的策略类交互
 * 这样的话，具体的算法和直接的客户端调用分离了，使得算法可以独立于客户端独立的变化。
 * 如果使用spring的依赖注入功能，还可以通过配置文件，动态的注入不同策略对象，动态的切换不同的算法.
 */
class Context {
    private Strategy strategy; //当前采用的算法对象

    //可以通过构造器来注入
    public Context(Strategy strategy) {
        super();
        this.strategy = strategy;
    }

    //可以通过set方法来注入
    public void setStrategy(Strategy strategy) {
```

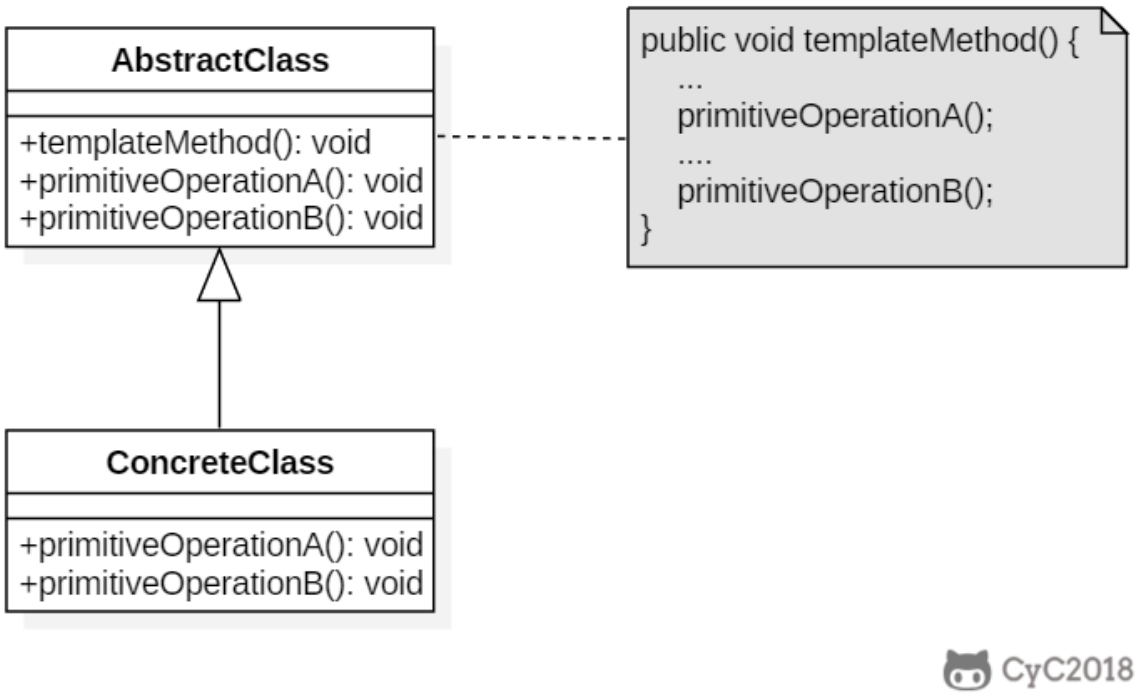
```
        this.strategy = strategy;
    }

    public void printPrice(double s) {
        System.out.println("您该报价: " + strategy.getPrice(s));
    }
}
```

模板方法（Template Method）

定义算法框架，并将一些步骤的实现延迟到子类。

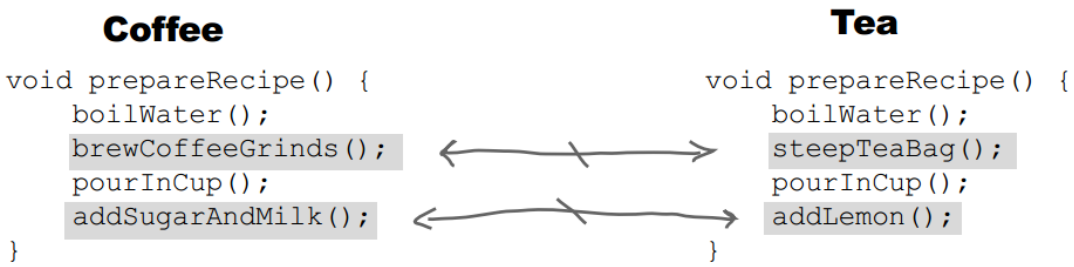
通过模板方法，子类可以重新定义算法的某些步骤，而不用改变算法的结构。

Class Diagram



Implementation

冲咖啡和冲茶都有类似的流程，但是某些步骤会有点不一样，要求复用那些相同步骤的代码。




```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
```

```
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("boilWater");
    }

    void pourInCup() {
        System.out.println("pourInCup");
    }
}
```

```
public class Coffee extends CaffeineBeverage {
    @Override
    void brew() {
        System.out.println("Coffee.brew");
    }

    @Override
    void addCondiments() {
        System.out.println("Coffee.addCondiments");
    }
}
```

```
public class Tea extends CaffeineBeverage {
    @Override
    void brew() {
        System.out.println("Tea.brew");
    }

    @Override
    void addCondiments() {
        System.out.println("Tea.addCondiments");
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        CaffeineBeverage caffeineBeverage = new Coffee();
        caffeineBeverage.prepareRecipe();
        System.out.println("-----------");
        caffeineBeverage = new Tea();
        caffeineBeverage.prepareRecipe();
    }
}
```

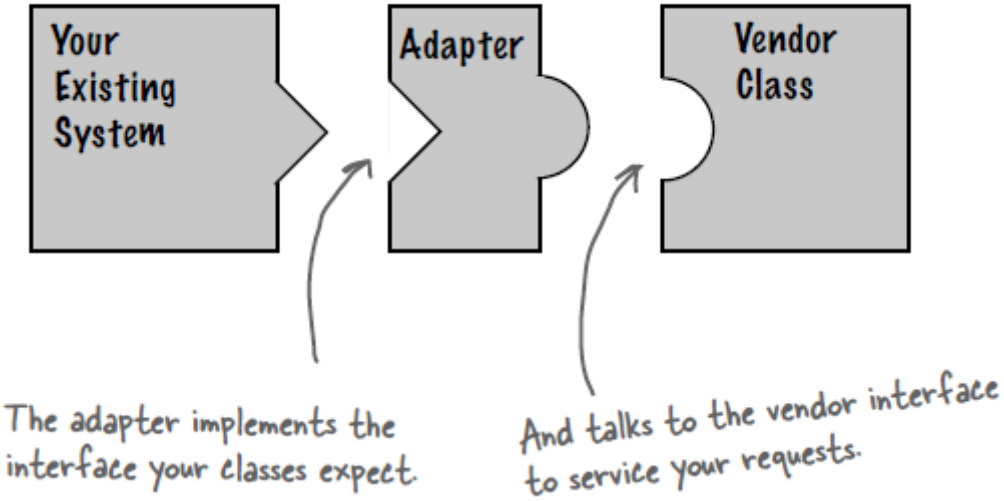
```
boilWater
Coffee.brew
pourInCup
Coffee.addCondiments
-----------
boilWater
Tea.brew
pourInCup
Tea.addCondiments
```

```
public class client {
    public static void main(String[] args) {
        BankTemplateMethod btm = new DrawMoney();
        btm.process();
        //采用匿名内部类
        BankTemplateMethod btm2 = new BankTemplateMethod() {
            @Override
            public void transact() {
                System.out.println("我要存钱");

            }
        };
        btm2.process();
    }
}


abstract class BankTemplateMethod {
    //具体方法
    public void takeNumber() {
        System.out.println("取号排队");
    }

    public abstract void transact(); //办理具体的业务 //钩子方法

    public void evaluate() {
        System.out.println("反馈评分");
    }

    /**
     * 模板方法:把基本操作组合一起，子类一般不能重写
     */
    public final void process() {
        this.takeNumber();
        this.transact();//像个钩子。执行时，挂哪个子类的方法就调用哪个
        this.evaluate();
    }
}

/**
 * 取款子类
 */
class DrawMoney extends BankTemplateMethod {
    @Override
    public void transact() {
        System.out.println("我要取款");
```

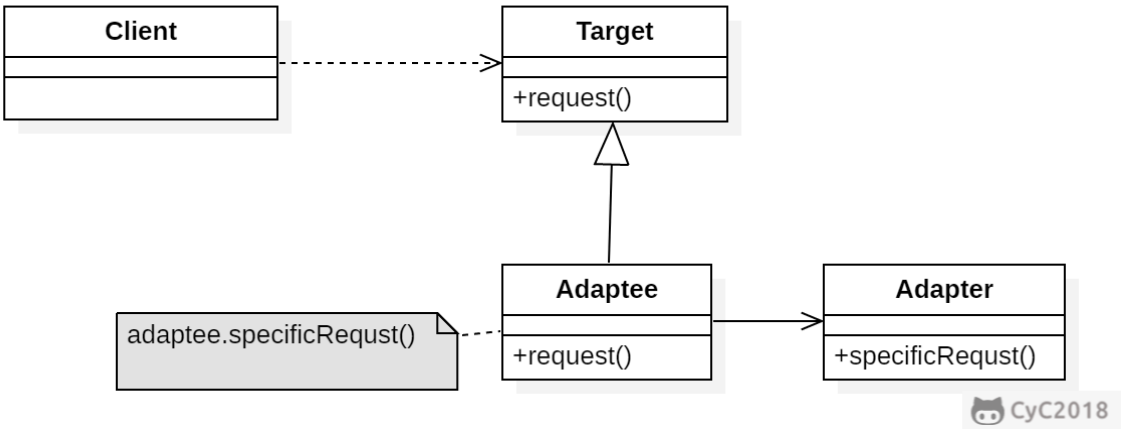
```
    }
}
```

适配器（Adapter）

把一个类接口转换成另一个用户需要的接口。



Class Diagram



Implementation

鸭子（Duck）和火鸡（Turkey）拥有不同的叫声，Duck 的叫声调用 quack() 方法，而 Turkey 调用 gobble() 方法。

要求将 Turkey 的 gobble() 方法适配成 Duck 的 quack() 方法，从而让火鸡冒充鸭子！

```
public interface Duck {
    void quack();
}
```

```
public interface Turkey {
    void gobble();
}
```

```
public class WildTurkey implements Turkey {
    @Override
    public void gobble() {
        System.out.println("gobble!");
    }
}
```

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    @Override
    public void quack() {
        turkey.gobble();
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Turkey turkey = new WildTurkey();
        Duck duck = new TurkeyAdapter(turkey);
        duck.quack();
    }
}
```

实现2：

条件：

1、有一个 *MediaPlayer* 接口和一个实现了 *MediaPlayer* 接口的实体类 *AudioPlayer*。默认情况下，*AudioPlayer* 可以播放 mp3 格式的音频文件。

2、有另一个接口 *AdvancedMediaPlayer* 和实现了 *AdvancedMediaPlayer* 接口的实体类。该类可以播放 vlc 和 mp4 格式的文件。

要求：让 *AudioPlayer* 播放其他格式的音频文件。

解决：创建一个实现了 *MediaPlayer* 接口的适配器类 *MediaAdapter*，并使用 *AdvancedMediaPlayer* 对象来播放所需的格式。

1.创建MediaPlayer、AdvancedMediaPlayer接口

2.创建AdvancedMediaPlayer实现类

3.创建MediaPlayer适配器类

4.创建MediaPlayer实体类并使用适配器类，使其可以播放其他格式的音频文件

```
public class AdapterPatternDemo {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.play("mp3", "beyond the horizon.mp3");
        audioPlayer.play("mp4", "alone.mp4");
```

```
        audioPlayer.play("vlc", "far far away.vlc");
        audioPlayer.play("avi", "mind me.avi");
    }
}

/**
 * 1.1创建媒体播放器接口
 */
interface MediaPlayer {
    public void play(String audioType, String fileName);
}

/**
 * 1.2创建高级媒体播放器接口
 */
interface AdvancedMediaPlayer {
    public void playVlc(String fileName);
    public void playMp4(String fileName);
}
/**
 * 2.1创建AdvancedMediaPlayer接口的实体类（播放vlc）。
 */
class VlcPlayer implements AdvancedMediaPlayer{
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file. Name: "+ fileName);
    }

    @Override
    public void playMp4(String fileName) {
        //什么也不做
    }
}

/**
 * 2.2创建AdvancedMediaPlayer接口的实体类（播放mp4）。
 */
class Mp4Player implements AdvancedMediaPlayer{

    @Override
    public void playVlc(String fileName) {
        //什么也不做
    }

    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: "+ fileName);
    }
}

/**
 * 3.创建MediaPlayer 接口的适配器类。
 */
class MediaAdapter implements MediaPlayer {

    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType){
```

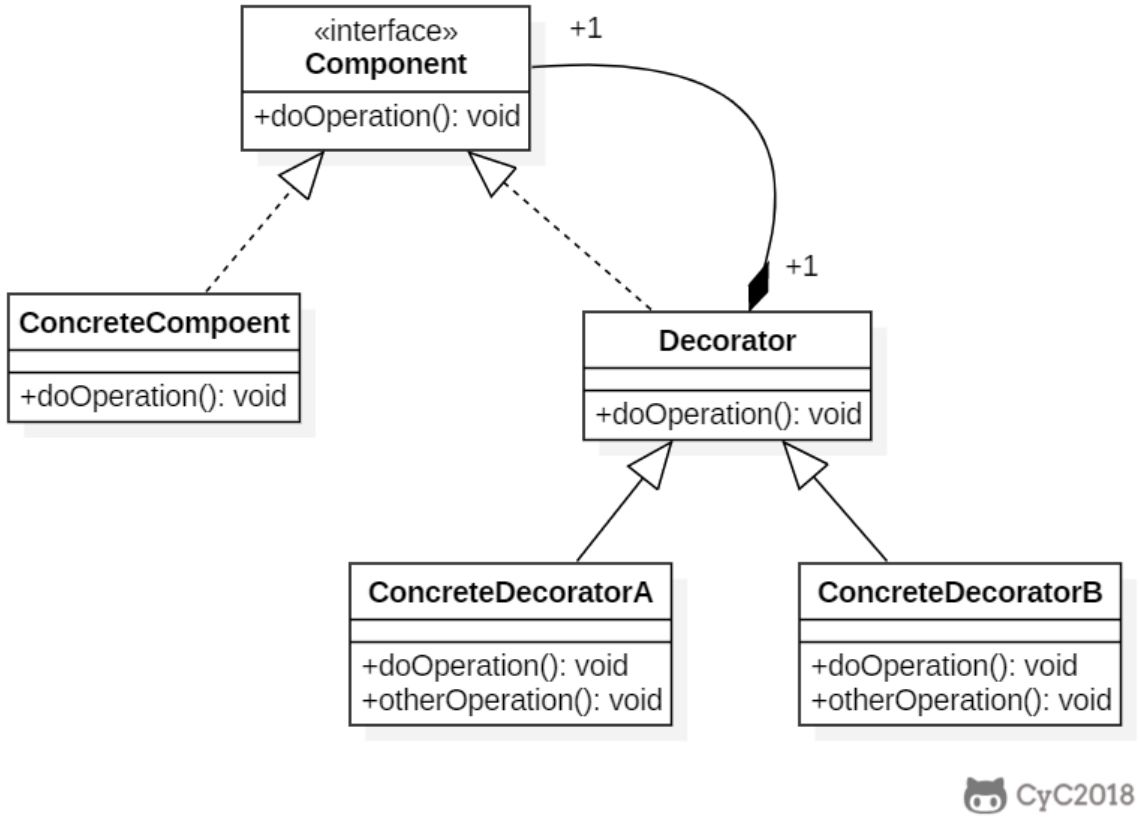
```
        if(audioType.equalsIgnoreCase("vlc") ){
            advancedMusicPlayer = new VlcPlayer();
        } else if (audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer = new Mp4Player();
        }
    }

    @Override
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("vlc")){
            advancedMusicPlayer.playVlc(fileName);
        }else if(audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}

/**
 * 4.创建MediaPlayer 接口的实体类
 * 让其可以播放其他格式的音频文件
 */
class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {

        //播放 mp3 音乐文件的内置支持
        if(audioType.equalsIgnoreCase("mp3")){
            System.out.println("Playing mp3 file. Name: "+ fileName);
        }
        //mediaAdapter 提供了播放其他文件格式的支持
        else if(audioType.equalsIgnoreCase("vlc")
                || audioType.equalsIgnoreCase("mp4")){
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        }
        else{
            System.out.println("Invalid media. "+
                    audioType + " format not supported");
        }
    }
}
```

装饰（Decorator）

为对象动态添加功能。

Class Diagram

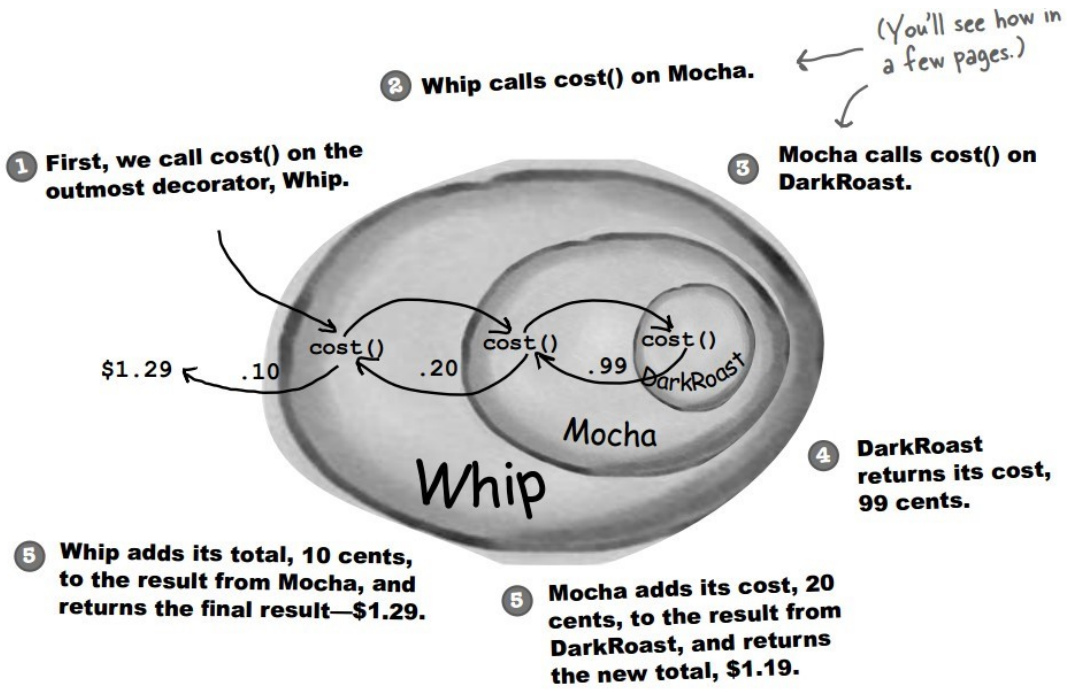
装饰者（Decorator）和具体组件（ConcreteComponent）都继承自组件（Component），具体组件的方法实现不需要依赖于其它对象，而装饰者组合了一个组件，这样它可以装饰其它装饰者或者具体组件。所谓装饰，就是把这个装饰者套在被装饰者之上，从而动态扩展被装饰者的功能。装饰者的方法有一部分是自己的，这属于它的功能，然后调用被装饰者的方法实现，从而也保留了被装饰者的功能。可以看到，具体组件应当是装饰层次的最低层，因为只有具体组件的方法实现不需要依赖于其它对象。



Implementation

设计不同种类的饮料，饮料可以添加配料，比如可以添加牛奶，并且支持动态添加新配料。每增加一种配料，该饮料的价格就会增加，要求计算一种饮料的价格。

下图表示在 DarkRoast 饮料上新增新添加 Mocha 配料，之后又添加了 Whip 配料。DarkRoast 被 Mocha 包裹，Mocha 又被 Whip 包裹。它们都继承自相同父类，都有 cost() 方法，外层类的 cost() 方法调用了内层类的 cost() 方法。




```
public interface Beverage {
    double cost();
}
```

```java
public class DarkRoast implements Beverage {
    @Override
    public double cost() {
        return 1;
    }
}
```

```java
public class HouseBlend implements Beverage {
    @Override
    public double cost() {
        return 1;
    }
}
```

```java
public abstract class CondimentDecorator implements Beverage {
    protected Beverage beverage;
}
```

```java
public class Milk extends CondimentDecorator {

    public Milk(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public double cost() {
        return 1 + beverage.cost();
    }
}
```

```java
public class Mocha extends CondimentDecorator {

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public double cost() {
        return 1 + beverage.cost();
    }
}
```

```java
public class Client {

    public static void main(String[] args) {
        Beverage beverage = new HouseBlend();
        beverage = new Mocha(beverage);
        beverage = new Milk(beverage);
        System.out.println(beverage.cost());
    }
}
```

设计原则

类应该对扩展开放，对修改关闭：也就是添加新功能时不需要修改代码。饮料可以动态添加新的配料，而不需要去修改饮料的代码。

不可能把所有的类设计成都满足这一原则，应当把该原则应用于最有可能发生改变的地方。

(静态)代理（Proxy）

为其他对象提供一种代理以控制对这个对象的访问。

注意事项：

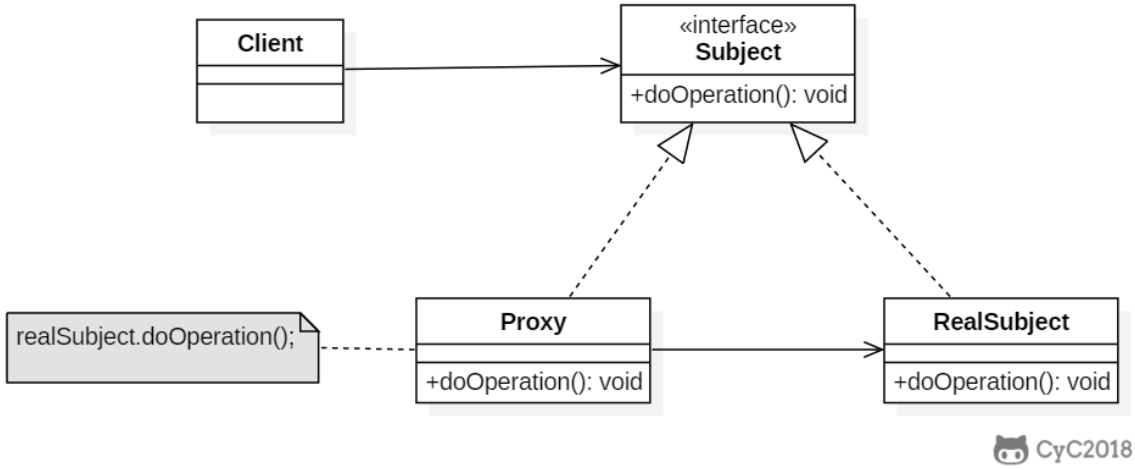
1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。

2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。

Class Diagram

代理有以下四类：

- 远程代理（Remote Proxy）：控制对远程对象（不同地址空间）的访问，它负责将请求及其参数进行编码，并向不同地址空间中的对象发送已经编码的请求。
- 虚拟代理（Virtual Proxy）：根据需要创建开销很大的对象，它可以缓存实体的附加信息，以便延迟对它的访问，例如在网站加载一个很大图片时，不能马上完成，可以用虚拟代理缓存图片的大小信息，然后生成一张临时图片代替原始图片。
- 保护代理（Protection Proxy）：按权限控制对象的访问，它负责检查调用者是否具有实现一个请求所必须的访问权限。
- 智能代理（Smart Reference）：取代了简单的指针，它在访问对象时执行一些附加操作：记录对象的引用次数；当第一次引用一个对象时，将它装入内存；在访问一个实际对象前，检查是否已经锁定了它，以确保其它对象不能改变它。



Implementation

1、真实角色

2、代理角色

3、两种角色实现相同的接口

代理相当于在代理角色中传入真实角色，对真实角色的方法前后做了一些处理。

模拟结婚：

```
public class StaticProxy {
    public static void main(String[] args) {
        new WeddingCompany(new You()).happyMarry();

        //new Thread(线程对象).srart();
    }
}

interface Marry {
    void happyMarry();
}

//真实角色
class You implements Marry {
    @Override
    public void happyMarry() {
        System.out.println("you and 嫦娥终于奔月了。。。");
    }
}

//代理角色
class WeddingCompany implements Marry {
    //真实角色
    private Marry target;

    public WeddingCompany(Marry target) {
        this.target = target;

    }

    @Override
    public void happyMarry() {
        ready();
        this.target.happyMarry();
        after();
    }

    private void ready() {
        System.out.println("布置猪窝。。。");
    }

    private void after() {
        System.out.println("闹玉兔。。。");
    }
}
```

以下是一个虚拟代理的实现，模拟了图片延迟加载的情况下使用与图片大小相等的临时内容去替换原始图片，直到图片加载完成才将图片显示出来。

```
public interface Image {
    void showImage();
}
```

```
public class HighResolutionImage implements Image {

    private URL imageURL;
```

```
    private long startTime;
    private int height;
    private int width;

    public int getHeight() {
        return height;
    }

    public int getWidth() {
        return width;
    }

    public HighResolutionImage(URL imageURL) {
        this.imageURL = imageURL;
        this.startTime = System.currentTimeMillis();
        this.width = 600;
        this.height = 600;
    }

    public boolean isLoad() {
        // 模拟图片加载，延迟 3s 加载完成
        long endTime = System.currentTimeMillis();
        return endTime - startTime > 3000;
    }

    @Override
    public void showImage() {
        System.out.println("Real Image: " + imageURL);
    }
}
```

```
public class ImageProxy implements Image {

    private HighResolutionImage highResolutionImage;

    public ImageProxy(HighResolutionImage highResolutionImage) {
        this.highResolutionImage = highResolutionImage;
    }

    @Override
    public void showImage() {
        while (!highResolutionImage.isLoad()) {
            try {
                System.out.println("Temp Image: " +
highResolutionImage.getWidth() + " " + highResolutionImage.getHeight());
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        highResolutionImage.showImage();
    }
}
```

```
public class ImageViewer {

    public static void main(String[] args) throws Exception {
        String image = "http://image.jpg";
        URL url = new URL(image);
        HighResolutionImage highResolutionImage = new HighResolutionImage(url);
        ImageProxy imageProxy = new ImageProxy(highResolutionImage);
        imageProxy.showImage();
    }
}
```

(动态)代理

1.JDK自带的动态代理

2.CGLIB

3.ASM(底层使用指令，可维护性较差)

动态生成代理类

* 特点：字节码随用随创建，随用随加载
  * 作用：不修改源码的基础上对方法增强
  * 分类：
  * 基于接口的动态代理
  * 基于子类的动态代理

JDK自带的动态代理

- java.lang.reflect.Proxy

  * 作用：动态生成代理类和对象

- java.lang.reflect.InvocationHandler

  * 可以通过invoke方法实现对真实角色的代理访问

  * 每次通过Proxy生成代理类对象时都要指定对应的处理器对象

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class client {
    public static void main(String[] args) {
        RealStar realStar = new RealStar();
        StarHandler starHandler = new StarHandler(realStar);
        Star proxy = (Star)
Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(), new Class[]
{Star.class}, starHandler);
        proxy.sing();
    }
}

interface Star {
    /**
     * 面谈
     */
```

```
    void confer();

    /**
     * 签合同
     */
    void signContract();

    /**
     * 订票
     */
    void bookTicket();

    /**
     * 唱歌
     */
    void sing();

    /**
     * 收钱
     */
    void collectMoney();
}

class RealStar implements Star {
    @Override
    public void confer() {
        System.out.println("RealStar面谈");
    }

    @Override
    public void signContract() {
        System.out.println("RealStar签合同");


    }

    @Override
    public void bookTicket() {
        System.out.println("RealStar订票");

    }

    @Override
    public void sing() {
        System.out.println("RealStar唱歌");

    }

    @Override
    public void collectMoney() {
        System.out.println("RealStar收钱");

    }
}

class StarHandler implements InvocationHandler {
    Star realStar;
```

```
    public StarHandler(Star realStar) {
        super();
        this.realStar = realStar;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        Object object = null;
        System.out.println("真正的方法执行前：");
        System.out.println("面谈，签合同，预付款，订票");
        if (method.getName().equals("sing")) {
            object = method.invoke(realStar, args);
        }
        System.out.println("真正的方法执行后：");
        System.out.println("收尾款");

        return object;
    }
}
```

基于接口的动态代理

* 基于接口的动态代理：

  * 涉及的类：Proxy
  * 提供者：JDK官方
* 如何创建代理对象：
  * 使用Proxy类中的newProxyInstance方法
  * 创建代理对象的要求：
  * 被代理类最少实现一个接口，如果没有则不能使用

* newProxyInstance方法的参数：
  * ClassLoader：类加载器
  *它是用于加载代理对象字节码的。和被代理对象使用相同的类加载器。写法：

    被代理对象.getClass().getClassLoader()
  * Class[]：字节码数组
  *它是用于让代理对象和被代理对象有相同方法。写法：

    被代理对象.getClass().getInterfaces()
  * InvocationHandler：用于提供增强的代码
  * 它是让我们写如何代理。我们一般都是些一个该接口的实现类，通常情况下都是匿名内部类，但不是必须的。写法：

```
new InvocationHandler() {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable{
        。。。
    }
}
```

  * 此接口的实现类都是谁用谁写。

接口

```
public interface IProducer {
    public void saleProduct(float money);
    public void afterService(float money);
}
```

实现类

```
public class Producer implements IProducer{
    public void saleProduct(float money){
        System.out.println("销售产品，并拿到钱："+money);
    }
    public void afterService(float money){
        System.out.println("提供售后服务，并拿到钱："+money);
    }
}
```

代理类

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class Client {
    public static void main(String[] args) {
        //匿名内部类访问外部成员变量时 要用final修饰
        final Producer producer = new Producer();

        IProducer proxyProducer = (IProducer) Proxy.newProxyInstance(
                producer.getClass().getClassLoader(),
                producer.getClass().getInterfaces(),
                new InvocationHandler() {
                    /**
                     * 作用：执行被代理对象的任何接口方法都会经过该方法
                     * 方法参数的含义
                     * @param proxy  代理对象的引用（如果在方法中想用代理对象可以用它，一
般不用）
                     * @param method  当前执行的方法
                     * @param args    当前执行方法所需的参数
                     * @return 和代理对象方法有相同的返回值
                     * @throws Throwable
                     */
                    @Override
                    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                        //提供增强的代码
                        Object returnValue = null;
                        //1.获取方法执行的参数
                        Float money = (Float) args[0];
                        //2.判断当前方法是不是销售
                        if ("saleProduct".equals(method.getName())) {
                            //被代理对象的引用，  和代理对象方法有相同的返回值
                            returnValue = method.invoke(producer, money * 0.8f);
                        }
                        return returnValue;
```

```
                    }
                });
        proxyProducer.saleProduct(10000f);
    }
}

interface IProducer {
    public void saleProduct(float money);

    public void afterService(float money);
}

class Producer implements IProducer {
    public void saleProduct(float money) {
        System.out.println("销售产品，并拿到钱：" + money);
    }

    public void afterService(float money) {
        System.out.println("提供售后服务，并拿到钱：" + money);
    }
}
```

基于子类的动态代理

* 基于子类的动态代理：
  * 涉及的类：Enhancer
  * 提供者：第三方cglib库，即需要导入jar包（maven工程需要导入依赖）

```
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>2.1_3</version>
</dependency>
```

* 如何创建代理对象：

```
*   使用Enhancer类中的create方法
*   创建代理对象的要求：被代理类不能是最终类
*   create方法的参数：
```

    * Class：字节码
    * 它是用于指定被代理对象的字节码。写法：

        被代理对象.getClass()

  * Callback：用于提供增强的代码
  * 它是让我们写如何代理。我们一般都是些一个该接口的实现类，通常情况下都是匿名内部类，但不是必须的。
  * 此接口的实现类都是谁用谁写。
  * 我们一般写的都是该接口的子接口实现类：MethodInterceptor

需要代理的类

```
public class Producer {
    public void saleProduct(float money){
        System.out.println("销售产品，并拿到钱："+money);
    }
    public void afterService(float money){
        System.out.println("提供售后服务，并拿到钱："+money);
    }
}
```

代理类

```
public class Client {
    public static void main(String[] args) {
        final Producer producer = new Producer();
        Producer cglibProducer = (Producer)Enhancer.create(
            producer.getClass(),
            new MethodInterceptor() {
                /**
                 * 执行被代理对象的任何方法都会经过该方法
                 * @param proxy
                 * @param method
                 * @param args
                 *   以上三个参数和基于接口的动态代理中invoke方法的参数是一样的
                 * @param methodProxy ：当前执行方法的代理对象
                 * @return
                 * @throws Throwable
                 */
                @Override
                public Object intercept(Object proxy, Method method, Object[]
args, MethodProxy methodProxy) throws Throwable {
                    //提供增强的代码
                    Object returnValue = null;
                    //1.获取方法执行的参数
                    Float money = (Float)args[0];
                    //2.判断当前方法是不是销售
                    if("saleProduct".equals(method.getName())) {
                        returnValue = method.invoke(producer, money*0.8f);
                    }
                    return returnValue;
                }
            });
        cglibProducer.saleProduct(12000f);
    }
}
```